The Performance of the Intel TFLOPS Supercomputer

Greg Henry, Enterprise Server Group, Beaverton, OR, Intel Corp.
Pat Fay, Enterprise Server Group, Beaverton, OR, Intel Corp.
Ben Cole, Enterprise Server Group, Beaverton, OR, Intel Corp.
Timothy G. Mattson, Microcomputer Research Laboratory, Hillsboro, OR, Intel Corp.

Index words: Parallel Supercomputer Applications

Abstract

The purpose of building a supercomputer is to provide superior performance on real applications. In this paper, we describe the performance of the Intel TFLOPS Supercomputer starting at the lowest level with a detailed investigation of the Pentium® Pro processor and the supporting memory subsystem. We follow this with a description of the benchmarks used to track the performance of the machine over its development life cycle, which culminated in the first MP LINPACK run to exceed a rate of one trillion floating point operations per second (TFLOPS). Our analysis applies not only to the TFLOPS supercomputer, but also to servers and workstations based on the Intel 32-bit architecture. We conclude with a discussion of the machine's performance on a production application.

Introduction

The Intel TFLOPS Supercomputer, also known as the ASCI Option Red Supercomputer, at Sandia National Laboratories in Albuquerque, NM, is the world's fastest supercomputer. By this we mean that this supercomputer is theoretically capable of doing more floating point operations per second on a given application than any other general purpose supercomputer built to date. With over 9200 Intel Pentium Pro processors each of which is capable of running at 200 million floating point operations per second (MFLOPS), this supercomputer can theoretically run at over 1.8 trillion floating point operations per second (TFLOPS).

An overview of what the supercomputer is and how it is used, the operating systems and parallel I/O running on it, and the scalable platform services that support it are the subjects of other papers in this Q1'98 issue of the Intel Technology Journal. This paper looks at how you achieve high performance with real applications. This improved performance cannot be achieved by adding more or faster nodes since the hardware is fixed. Therefore, we look at algorithmic and coding enhancements to the applications. Furthermore, we investigate what kinds of performance can be reasonably expected, and what can be done to enhance the performance of given applications.

It can be argued that the first barrier to achieving performance on an application is parallelizing it. That is, the data and/or work must be efficiently distributed amongst all the processors in order to achieve optimum performance from the processors working together. How easy, hard, or possible this is depends on the application. It is not our goal here to discuss this difficulty. As mentioned in another paper in this issue, there are around 4500 compute nodes in this supercomputer each having two processors. Let us assume that an application can be at least distributed among these 4500 nodes. Since the funding for this supercomputer comes from the DOE–an organization with vast experience in scalable computing–assuming that the application is parallelized is not entirely unreasonable. And although none of the applications have ever been run on such a large parallel supercomputer, the scientists at Sandia National Laboratories have spent many years achieving parallelism in their data and know how to take advantage of a scalable supercomputer.

Hence, instead of discussing application parallelization, we discuss the efforts required to achieve high performance of existing parallel applications. The total efficiency of the full system cannot be better than the efficiency of a single node. Much of our discussion is focused on a single node or even a single processor. We start with a quick introduction to the Pentium Pro processor followed by our initial performance explorations on a processor and its supporting memory subsystem. We then explore some of the benchmarks used to track system performance and discuss our historic MP LINPACK computation. The paper concludes with a brief discussion of a specific application called CTH.

The Pentium Pro® Processor

The Intel Pentium Pro processor is used on all the nodes in the Intel ASCI Option Red Supercomputer. A full description of the processor is beyond the scope of this paper and is available elsewhere [17]. In this section, we highlight the key features of the processor and emphasize issues that are important when analyzing application performance.

At runtime, an instruction for the Pentium Pro processor is broken down into simpler instructions called micro-operations (or *uops*). Three decode units are available on the processor to carry out this decomposition. One unit (unit 0) can decode complex operations while any of the three units can decode simple operations. The uops execute inside the Pentium Pro processor with the order of execution dictated by the availability of data. This lets the CPU continue with productive work when other uops are waiting for data or functional units. This *out of order execution* is combined with sophisticated *branch prediction* and *register renaming* to provide what Intel calls, *dynamic execution*.

The Pentium Pro processor's core can execute a burst rate of up to five uops per cycle running on five functional units:

- Store Data Unit
- Store Address Unit
- Load Address Unit
- Integer ALU
- Floating Point/Integer Unit

Up to three uops can be retired per cycle of which only one can be a floating-point operation. The floating-point unit requires two cycles per multiply and one cycle per add. The adds can be interleaved with the multiplies so the Pentium Pro processor can have a result ready to retire every cycle. Hence, the peak multiply-add rate is 200 MFLOPS at 200 MHz.

The Pentium Pro processor has separate on-chip data and instruction L1 caches (each of which is eight KBytes). It also has an L2 cache (256 KBytes) packaged with the CPU in a single dual-cavity PGA package. Cache lines are 32 bytes wide. The L1 data cache is dual-ported and non-blocking, supporting one load and one store per cycle for peak bandwidth of 3.2 billion bytes per second (GB/sec) on a 200 MHz CPU. The L2 cache interface runs at the full CPU clock speed and can transfer 64 bits per cycle (1.6 GB/sec on a 200 MHz Pentium Pro processor). The external bus is also 64 bits wide and supports a data transfer every bus-cycle.

The Pentium Pro processor bus offers full support for memory and cache coherency for up to four Pentium Pro processors (though our compute nodes only have two processors). It has 36 bits of address and 64 bits of data. Bus efficiency is enhanced through the following features:

- the capability to defer long transactions
- a bus pipeline with a depth of 8
- bus arbitration on a cycle-by-cycle basis

The bus can support up to eight pending transactions while the Pentium Pro processor and the memory controller can have up to four pending transactions. Memory controllers can be paired-up to match the bus's support for eight pending transactions.

The bus can sustain data on every clock cycle, so at 66 MHz, the peak data rate is 533 million bytes per second. Unlike most CCOTS processor buses, which can only detect data errors by using parity coverage, the Pentium Pro processor data bus is protected by ECC. Address signals are protected by parity.

Memory Movement

The first obstacle to fast parallel performance is the per node performance. Therefore, in this section, we discuss how fast we were able to run code on just one node, which has two 200 MHz Pentium Pro processors. (In many cases, our observations could be extended to the Intel Pentium® II processors, with appropriate adjustments made in the core frequency and cache specifications.)

Scientific applications tend to differ from many commercial applications in that they often have huge data sets, and large amounts of floating point operations are done on these data sets. How quickly data can be moved and manipulated is significant.

Let us start with our definition of memory movement. We use the phrase "memory movement" to refer to the movement of data into the floating-point stack or into the L1_data cache. Frequently, memory movement discussions are restricted to the movement of data from main memory. However, we needed to focus on the larger issue of using data, whether it might already be in cache or not. Therefore, issues such as the instruction decoding sequence and how it affects the rate that data can be pulled from the L1_data cache are included in this discussion of memory movement. In essence, our focus is on anything that impacts memory movement.

Studying memory movement leads to an investigation of the concept of hierarchical memory. There are a finite number of resources available in the hierarchy and any memory reference must eventually traverse the entire hierarchy. As an example, if you do a store of a value in a register to main memory, you need to go to L1 and L2 on your way to main memory, although the write-back nature of the caches may prevent this from happening immediately. The fastest resources (the registers) can work with the smallest amount of data. There are levels of cache to improve data movement efficiency built on top of

the registers. The further one goes from the registers, the longer the wait and the slower the bandwidth for memory movement, as well as the more data it can hold. The base of this pyramid is often the main memory, which in our case is 128 Mbytes (1 Mbyte = 2^{20} bytes), although this hierarchical scheme can easily be extended to include multiple nodes, and finally the parallel I/O subsystem. (As mentioned previously, we choose to focus our attention in this section on the single node model, which includes the registers through the main memory subsystem.)

Despite having 40 internal floating-point registers, the application program can only address the 8 floating point register stack. In many cases, this was a significant bottleneck to overall performance.

The next level of memory hierarchy is the primary on-chip non-blocking L1_data and instruction cache. For the Pentium Pro processor, these are 8 Kbytes each and are two-way set associative, write-back and write-allocate. Reading data from L1 can be done at 1600 million bytes per second or 1526 Mbytes/sec, which amounts to one double per CPU cycle. Writing data can also be done at the same speed. Furthermore, two CPUs can be simultaneously reading and writing data to/from their perspective L1 caches. We prefer, however, to simplify the discussion by temporarily ignoring these issues. More to the point, we claim it is rare that a scientific application will have many reads and writes to the L1_data cache occurring at the same time.

The next level of memory hierarchy is the non-blocking unified off-die L2 cache. In our case, the L2 cache holds 256 Kbytes and can theoretically sustain 1600 million bytes per second reading exclusive or writing. All cache lines are 32-bytes wide.

A compute node has a memory subsystem with 128 Mbytes. The PCI bus operates at 33 MHz. The memory bus runs at 66 MHz., is 64 bits wide, and can support a data transfer every bus cycle. That is, it can do one 8 byte (64 bit) transaction every bus clock. This amounts to a bandwidth of an 8 byte read or write to L2 every 3 cpu clocks or 1600/3 = 533 million bytes/sec (508 Mbytes/sec.)

The first step in our investigation was to find out how fast we could do simple floating-point operations from various levels of memory. First, we looked at how quickly we could load and pop the data (from various places on the hierarchy) onto the floating-point stack. Second, we looked at how we could use this information to build simple kernels like element vector multiply (x(i) = y(i)*z(i), i= 1,...,n) or matrix-matrix multiply (for MP LINPACK).

It is important here to emphasize the specific nature of our study. Other research papers and/or projects tend to look no further than how fast memory can be moved. But for us, since our final goal is ultimately floating point calculations, by necessity our investigation targeted the floating-point stack. IA-32 instructions like REP MOVS (repeat move string long), although interesting for timing memory movement, are not sufficient to meet our final goal. Following is a discussion of those results and observations that bear on the benchmarks and applications discussed later in this paper.

Overheads Due to the Instruction Decoder

There are three instruction decoders on the Intel Pentium Pro processor. These decode complicated IA-32 instructions into simple uops. Only the processor uses the uops: a programmer can not directly code in terms of uops. Only one of the decoders can decode "complex" instructions. The following problems are directly based on overheads due to the decoder. While none of these problems are measurable when moving data from memory, they can be observed when we know the data lies in the L1_data cache.

Reading data from the L1_data cache onto the floating point stack has a theoretical upper bound of one double per clock or 1600 million bytes a second. Using integer touches like "movl 8(%eax), %ebp," as opposed to floating point touches like "fldl 8(%eax)," we have come within a percent of this speed. However, using floating point loads and pops (we say pops instead of stores because the typical case is that we have many loads followed by add-pops or mul-pops and then perhaps only one store), we observed stalls every time an instruction passed over a 16-byte instruction boundary. A random sampling of floating point showed that floating point instructions are typically 5-7 bytes in length. This means that typical compiled floating point codes tend to run at only 80 percent efficiency out of the L1_data cache, encountering a stall every fourth instruction.

Our best code, which tried to avoid this problem, peaked at 1450 Mbytes/sec out of a possible 1526 Mbytes/sec. To achieve this level of performance, we had to use a potentially unrealistic degree of loop unrolling. We also had to keep the offsets small so that the instructions could be decoded in a smaller number of bytes. (A floating point load off of a location in a register can be a two-byte instruction if there is no offset, a 3-byte instruction for offsets up to 128 bytes, and a 6-byte instruction or more for larger offsets.) Another way of viewing this is if you know that you have X data loads from L1_data, instead of taking minimally X cycles, it is likely to take 1.25*X cycles.

These observations pointed out a critical performance issue: codes tend to run faster when the instructions are simpler. Complicated instructions can lead to a stall because only decoder 0 can decode the complicated instructions. Things like multiplying an element from

memory with an element in floating point stack location 0 can be implemented with a single IA-32 instruction, but require several different micro-operations to execute. Codes may often be faster if they are implemented with simpler instructions. For our example, one could first do a load and then in the second instruction do the multiply.

Floating Point Registers and Pipelining

When moving data from L2 or main memory, we often found that we needed to touch several different cache lines in order for the pipeline to be deep enough to obtain the faster bandwidths necessary for scientific calculations. This was one of the cases where having too few floating point registers was immediately apparent. In some cases, we sped up codes by interleaving integer touches of cache-lines before we actually did the explicit floating-point load onto the stack. This was critical to performance tunings of matrix multiply for MP LINPACK, for example.

An important finding was that a single 200 MHz Pentium Pro processor cannot saturate the memory bus bandwidth. That is, out of the 533 million bytes a second or 508 Mbytes/sec, the fastest bandwidth we achieved was around 428 Mbytes/sec or 85 percent of the maximum theoretical bandwidth. Two 200 MHz CPUs, on the other hand, achieved 491 Mbytes/sec. Several tricks were used to achieve this higher performance. We needed to synchronize the CPUs at critical points using a special utility we created. We needed both CPUs to saturate the bus (although higher frequency Pentium II processors are more likely to saturate their bus). We also needed to unroll our floating point calculations sufficiently enough to ensure that there were always several outstanding cache-line requests at once, such as touching doubles X(1), then X(5), then X(9), etc., before attempting to access X(2). This enabled the pipeline to remain busy. Using a second processor to access data from main memory tended to yield two benefits: not only could we then issue instructions fast enough to saturate the bus, but we could also have a second set of eight floating point stack locations by which we could unroll things. Because there were two benefits, this enabled some memory-bound codes to enjoy greater than the 15 percent improvement possible from just saturating the bus. In fact, some memory-bound codes when run on two CPUs actually achieved around a 30 percent benefit. (Naturally, cache-bound codes often achieved a 2x improvement.)

Unfortunately, using two CPUs is sometimes a double-edged sword: accessing different DRAM pages always caused an expensive stall. Although this can and often does happen with a single CPU, it happens far more readily with two CPUs. When using the second processor, the best method is to access alternate cache lines in the same DRAM page. This will allow codes to make effective use of 16 floating point stack locations instead of just 8, and will also prevent the thrashing of memory (page miss penalties) if the different CPUs are working on entirely different DRAM pages. Another big benefit of alternating cache lines is that each processor avoids having to snoop modified cache lines from the other processor's caches.

The difficulty with studying simple kernels is that the critical cases that actually occur in practice are sometimes overlooked. We found our simple examples of moving data from a single vector in L2 or main memory onto the floating-point stack to be insufficient. We therefore launched a study involving the movement of two vectors at the same time, which we found to be simple enough to optimize and realistic enough to capture the major bottlenecks.

Using our new set of kernels, we made several interesting observations. For starters, we found that accessing data from L2 onto the floating point stack tended to run at 800 Mbytes/sec in this somewhat more realistic mode. We found this a little disappointing since it represented only just over half the bandwidth theoretically possible. We also studied the impact of touching one vector from main memory and another from L1. This has a theoretical bandwidth of 16 / ((8/1526)+(8/508)) = 763 Mbytes/sec; however, it was often harder to achieve more than 530 Mbytes/sec. This implied not only that there was no overlap when loading from the two separate places but that the two loads interfered with one another. For touching two vectors from memory, we found that pre-touching at least one of them first with integer touches in small enough chunks to keep the data in L1 allowed performances up to 320 Mbytes/sec out of a possible 508.

We have illustrated how using the registers effectively can improve performance. There are several cases where simply not having enough registers hurts performance. The application CTH, illustrated in greater depth later in this paper, is a case where the same code and same data set tends to produce more loads on IA-32 than other architectures. Another example is an application called MPSalsa, which was a Gordon Bell (fastest real application running on a supercomputer) finalist at IEEE's Supercomputing 1997. At the core kernel, there were a series of matrix vector products from memory. Since the size of the vectors were small (around 24), they should have been able to fit into floating point registers. Instead, there was a significant overhead introduced by interleaving loads from memory with loads from L1 as described above.

Putting It All Together

We concentrated on a single floating point kernel called *element vector multiply* (EVM). This kernel sets X(i) = Y(i)*Z(i) as i goes from one to one million with double-precision vectors X, Y, and Z. Since this involves 24 million bytes of data, it is clearly a problem coming from main memory. Note that the write-back, write-allocate nature of L2 suggests that each operation involves loading X(i), Y(i), and Z(i), and then storing X(i). This is 4 doubles moved from main memory, which theoretically could be done in 4*3 CPU cycles, or one flop done every 12 CPU cycles. On a 200 MHz processor, that would suggest 16 MFLOPS or so. However, observed performance was 4 or 5 MFLOPS. We then set out to determine where the loss of performance was going given what we learned about simple memory movement kernels.

DRAM page misses are just one slowdown. The read of Y(i) followed by the read of Z(i) causes a page miss that halves bandwidth. We also have to read X(i) due to the write-allocate write-back memory mode. This causes another page fault. In write-back mode, data is only written back to memory to make room for something else to be brought into cache. This is eviction: one cannot control when data will be evicted. There is a penalty for intermixing reads and writes to memory. Recall that the 200 MHz Pentium Pro processor cannot keep the memory controller completely busy, thus utilizing 85% of the memory bandwidth.

We then set out to try to improve the speed. We blocked the loop such that we streamed in about 4K of the vector Y (about half of L1_data). This avoids the page faults that we get for alternating reads of X, Y, and Z. Then we loaded the vector Z into the other half of L1 (getting only one page fault for the initial read of Z). We then performed the multiply X(i)=Y(i)*Z(i) over the elements loaded into L1. This results in reading X(i) from memory due to the write-allocate memory model. The result X(i) is written to cache. We repeated this touching Y, touching Z, writing X until about half of L2 was filled with X's modified lines. Now if we were to continue we would start to evict lines of X as we read in Y and Z. This would cause page faults and drop performance back down. Once about half L2 is filled with X, we could do a Cougar instruction called flush_cache (which does the protected assembler WBINVD instruction) to write the modified data in the caches back to memory.

This blocking algorithm looks like:

```
for(sizeof(X/128K))
{
      for(128K of X)
      {
            touch 4K of Y
            touch 4k of Z
            calculate 4k of X=Y*Z
      }
      flush_cache()
}
```

The touch of Y runs at about 425 million bytes/sec. The touch of Z runs at about 390 million bytes/sec due to having to evict the dirty lines of X from L1 to L2. The multiply of 4k of X=Y*Z runs at about 200 million bytes/sec. It was a bit mysterious that this performance was not higher. The flush cache runs at 250 million bytes/sec peak (our own versions of flush cache appeared to run no faster).

The total throughput is then:

```
1/(1/425 + 1/390 + 1/200 + 1/250)
= 71.85 Mbytes/sec
```

written to X. The flops/sec is then 71.85/8 = 8.98 MFLOPS. This is still not close to the 16.6 MFLOPS, but it is 50% better than the naive loop.

The final set of experiments we made was on write-combine caching. Write-combined memory avoids reading the data before you write it. Also, when writing a whole cache-line, the write is "combined" and sent to the memory bus as one request. Note that write-combine doesn't read/write to the caches; rather, data is transferred directly to/from memory so cache coherency issues have to be addressed. Coherency can be handled by flushing cache at the beginning of the EVM routine and, if necessary, at the end of it also. We achieved about 16.8 MFLOPS with stride 1 write-combine EVM.

The write-combine memory model appears to be useful in kernels that involve writing large pieces (greater than L2 size) of contiguous data to memory. Using write-combine on a Pentium Pro processor-based system proved somewhat challenging; however, it is our untested understanding that the methodology is much easier on Pentium II processor-based workstations. If this is the case, then a great deal of our efforts can be applied to Pentium II processor-based platforms, thus enabling many users around the world to take advantage of our work.

Hardware Counters

When running applications on the Intel ASCI Option Red Supercomputer, it is often useful to know what portion of the data is running from what portion of memory. On a Windows NT box, a utility like Intel VTune[5] might find this information. However, in our operating system environment, more closely resembling UNIX, this was not an option, especially on applications

too large to fit on a single workstation. Therefore, we accessed the hardware counters directly. The ones we found most useful for studying memory movement were PP_DATA_MEM_REFS (0x43), PP_L2_LINES_IN (0x24), and PP_DCU_LINES_IN (0x45). Assuming that the number of references per element in every cache line accessed was the same, reads and writes rarely overlapped, and that the vast majority of data references were all double precision loads to the floating point stack, we generated the following observations:

The fraction of data from Memory, FracM, is

$$FracM = \frac{MIN(PP\_L2\_LINES\_IN * 4, PP\_DATA\_MEM\_REFS)}{PP\_DATA\_MEM\_REFS}$$

The number of L2 and L1 hits is

PP_DATA_MEM_REFS-PP_L2_LINES_IN*4,

and the number of just L2 hits is
MIN(ABS(PP_DCU_LINES_IN - PP_L2_LINES_IN)*4, PP_DATA_MEM_REFS)

The fraction of data from L2 is then

$$FracL2 = \frac{(Number\_of\_L2\_hits) * (1.0 - FracM)}{(Number\_of\_L2\_hits - Number\_of\_L1\_hits)}$$

The fraction of data from L1 is then
FracL1 = 1.0 - FracM - FracL2.

While the foregoing assumptions are simple and do not always apply, they gave us a useful estimate to work with. We could then apply this estimate to our observations about the overheads incurred when accessing data from the various levels of memory resulting in overall performance estimates for an application. For example, we discussed earlier that five cycles are typically used to access four elements from L1, so that the minimum number of cycles for accessing the data that was in L1 might be 1.25 * FracL1. Similarly, fudge factors of 1.59 existed for FracL2, and 3.57 (or 3.13 if dual processor) for FracM. An example of a situation where we used this is shown in the discussion on the application CTH.

Performance Tracking

Most of the early applications work on the Intel ASCI Option Red Supercomputer was designed to validate the soundness of the system design and its ability to scale to thousands of nodes. This work was quite successful with several applications (including some full production applications) running on up to 4500 nodes.

While it is important that the ASCI Option Red Supercomputer functions correctly, it is equally important that the system delivers the expected performance. To track system performance, we created a performance benchmark suite. The goal of this suite was to produce a handful of numbers to assess system performance. The

System	si238	si58	babyflop
Software Release	WW34a_1	WW45	1.2 WW39
Date tests were ran	9/17/96	12/31/96	12/4/97
Livermore Loops			
AM MFLOPS	33.9	42.6	48.3
GM MFLOPS	29.6	33.9	38.9
HM MFLOPS	24.3	25.9	29.1
Minimum MFLOPS	5.9	5.7	6.0
Maximum MFLOPS	61.4	111.8	118.4
Standard Deviation MFLOPS	16	28.4	29.4
Comtest			
Bandwidth - MBytes/sec	272.4	302	302
CSEND Latency - υsecs	12	10	9
Stream Test			
Copy MBytes/sec	85.9	109.1	114.9
Scale MBytes/sec	107.2	108.6	108.7
Add MBytes/sec	128.7	129	130.0
Triad MBytes/sec	128.5	129.3	129.4
Matrix Multiply			
F77, per-node MFLOPS	62.4	54.6	55.3
libkmath, per-node MFLOPS	119.4	111.1	112.3

Table 1: Results from the performance tracking benchmark suite. The tests are not strongly dependent on the number of nodes. These particular tests used four nodes. None of these tests used the second processor for computation. The System names refer to internal systems at Intel.

performance tracking suite includes the following codes:

Livermore Loops: A measure of the performance of the Fortran77 compiler with loops typical to scientific computing. The arithmetic (AM), geometric (GM), and harmonic means (HM) are reported as well as the range and standard deviation in the MFLOPS.

Comtest: Measures the bandwidth, latency, and standard deviation for a pair-wise, nearest neighbor ping-pong test.

McCalpin Stream: Measures performance of memory intensive applications [9]. Specific tests are vector copy, element-wise scale and add, and the triad (i.e., a(i)=a(i)+b(i)*c(i)).

Parallel Matrix Multiply: Measures performance of a parallel matrix multiply. The performance per node is reported in MFLOPS for a 4-node multiplication of order 300 matrices.

The performance levels are reported in Table 1 for several dates spread out over the course of the project. The numbers have largely stabilized, and significant additional improvements are not anticipated. The Livermore Loop and Stream test numbers are in the same ballpark as those from other high-end workstations. The communication numbers are among the best ever reported for an MPP system. Finally, the matrix multiplication numbers provide a measure of compiler performance by comparing MFLOPS rates for compiled and assembly-coded multiplications. The compiled code is a factor of two slower than the assembly code, which is not unusual compared to Fortran compilers on other high-end workstations.

These tests provide a good relative measure of the system performance. They are not very good, however, at detecting systematic errors in the system's performance. To resolve this issue, we needed a benchmark for which we have an analytic performance target. If we match this target, then we know our system is performing as it should.

An application well suited to this type of analysis is MP Quest [13], an ab initio quantum chemistry program developed at the Sandia National Laboratories. In an earlier study[10], we analyzed the nboxcd() kernel from MP Quest. This kernel resembles a modified dense matrix multiply operation. Our analysis showed that this kernel should run somewhere between 110 MFLOPS to 130 MFLOPS (depending on the state of the L2 cache prior to the kernel's operation).

We created a stand-alone benchmark program based on this kernel. Table 2 compares results for these tests built with the PGI compiler and the Intel C/C++ Compiler for Win32*systems. Three different releases of the PGI compilers are included: 9/96 (release 1.1), 12/96 (release 1.2-5), and the 12/97 (release 1.6-3). The Intel C/C++ compiler (9/96 release) is the Pentium Pro processor reference compiler developed by Intel. These single node computations were carried out on a 200 MHz -based node. These tests used two forms for the benchmark: one with the original code and the other with the loops unrolled. The expected optimum performance ranges are from 110-120 MFLOPS.

The Intel C/C++ compiler hits the target performance. This compiler is highly optimized for the Pentium Pro processor so its high performance is not surprising. The PGI compilers are well short of the target performance. (PGI is still working on the compiler, however, and future releases will hopefully close the gap.)

MP LINPACK Performance

MP LINPACK is a well known benchmark for high performance computing. The benchmark measures the time it takes to solve a real double precision (64 bits) linear system of equations with a single right-hand side. On December 4, 1996, we set a new world record for MP LINPACK by running the benchmark in excess of one TFLOPS. At that time, the Intel ASCI Option Red Supercomputer was only 80% complete, but that was more than enough to break the MP LINPACK TFLOPS barrier. Actually, the previous record was 368 GFLOPS so we did not just break the record, we shattered it!

While the rules for the LINPACK benchmark require use of the standard benchmark code, MP LINPACK lets you rewrite the program as long as certain ground rules are followed [6]. Our MP LINPACK code used a two-dimensional block scattered data decomposition with a block size 64 [9]. The algorithm is a variant of the right looking LU factorization with row pivoting and is done in

Code	PGI 9/96 (Rel 1.1)	PGI 12/96 (Rel 1.2-5)	PGI 12/97 (Rel 1.6-3)	Intel C/C++ Compiler
Original Kernel	26	56	67	83
Kernel with unrolled loops	30	75	87	120

Table 2: Performance in MFLOPS for the NBOXCD() Kernel from MP Quest.

accordance with LAPACK [1]. The parallel implementation [4,8,15] used a two-dimensional processor mesh and did a block wrapped mapping of the matrix. Columns of processors cooperated synchronously to compute a block of pivots that were then passed asynchronously across the rows. A look ahead pivot was used to keep pivoting out of the critical latency path. We report timings for real floating point operations and not "macho" FLOPS obtained by using Strassen [14] (or Winograd [16]) multiplication. The code explicitly computed all the relevant norms and did several rigorous residual checks to guarantee accuracy. The matrix generation was identical to ScaLAPACK version 1.00 Beta, which is a standard MPP package for Linear Algebra [2].

The benchmark results are maintained in the LINPACK Performance Report: "Performance of Various Computers Using Standard Linear Equations Software" by Dr. Jack Dongarra at the University of Tennessee [6]. He has accepted our TFLOPS entry into his 12/16/96 report, which is available on the web [6], e-mail, and ftp. RMAX was 1.068 TFLOP, NMAX or N was 215000, and N1/2 was 53400. N1/2 is the minimum problem size (to the nearest 100) such that half the RMAX performance was achieved. That is, over half a TFLOP was achieved on this machine using a problem size of 53400. The RMAX was found on 12/4/96, and N1/2 was found on 12/6/96. The number of floating point operations done is roughly (2N^3)/3 for a problem of size N.

The MP LINPACK 1.3 TFLOPS run (on 6/9/97) was run on 9152 Pentium Pro (TM) 200 MHz processors. RMAX was 1.338 TFLOPS. NMAX or N was 235000. N1/2 was 63000. Both runs used MPI.

The code for the 1.06 TFLOPS MP LINPACK record was derived from programs used to set earlier MP LINPACK records on Intel's Paragon supercomputers. The initial implementation was based on work by Robert van de Geijn [15]. The Delta code was modified to run on the Intel MP Paragon and it used hand-tuned Intel i860 processor assembly code kernels. For the TFLOPS benchmark, these kernels were written in x86 assembly code. For a detailed description of the techniques and algorithms used in this code, see the paper by Bolen et. al. [4]. Our past work with MP LINPACK has shown that for very large problems, at least 93% of the runtime is consumed by the BLAS-3 matrix multiplication code, DGEMM (which computes C=C-A*B). The dual processor code for large DGEMM problems ran at 345 MFLOPS.

Increasing Parallel Efficiency

We employ many techniques to increase parallel efficiency once a code has already been initially scaled. For MP LINPACK, we used the lookahead pivot technique described above. We also used a common optimization technique based on the observation that memory-to-memory copies tend to run at very slow speeds like 80 Mbytes/sec (see our previous results on element vector multiply) but the communication bandwidth of the machine is closer to 400 Mbytes/sec. This means that if an incoming message needs to be copied from an operating system into a user buffer, this takes more time than sending the message. When one posts a message ahead of time, and sends a "handshake" to tell a node it is ready to receive the message (a delicate process since we don't wish to introduce bottlenecks), the communication overheads go down, which enables a code to scale to more nodes. Sometimes it is even faster for a node to send a message to itself, than to call memcpy().

In some cases, we have to reduce I/O to assist in scalability. This is beyond the scope of this paper.

Matrix-Matrix Multiplication

The matrix-matrix multiplication behind MP LINPACK is an upper product update of the form C = C - A*B where C is large with usually slightly more rows than columns, and the number of columns of A (and rows of B) is typically small (in our case 64). Disregarding notations contained elsewhere, suppose C is MxN, A is MxK, and B is KxN, where M>=N>>K.

DGEMM has 2*M*N*K flops and at least 2*M*N + M*K + K*N memory references. If K is sufficiently large, cache re-use will be higher, and the loading and storing of C will be amortized. We typically block A into chunks that fit into L1, and B into chunks that fit into L2, and then complete the relevant portion of C before proceeding to the next chunk of A or B. Because L1_data is 8 Kbytes and 2-way set associative, we have found that it is unwise to use more than 4K of data for A in any one given time. For K=64, solving for 8 rows of C by copying 8 rows of A into a scratch space and doing the multiply is ideal for several reasons[8]. First, this means that 8*64*8 = 4096 bytes of A will hopefully remain L1_data cache resident. Since there is no convenient instruction for accessing across a row, and we would prefer to avoid continually updating the integer registers, copying A into a contiguous space helps side-step this problem because we can change the storage format of A. A DGEMM implementation on top of this is also beneficial because the issue of A or A transpose (another DGEMM option) becomes irrelevant since we always assume a copy of A[7]. Furthermore, we would prefer to process a number of rows that are a multiple of the cache line size to avoid additional cache movements when the initial arrays are aligned on cache-line boundaries.

An outer-level blocking outside the row blocking is done on the columns of B and C so that B always remains in L2. For unfavorable leading dimensions of B, another copy can be done on B. However, this can be avoided

within the context of a careful MP LINPACK implementation.

Due to a limitation of the number of floating point registers, the actual inner DGEMM kernel can only access a column of B or C at a time. Furthermore, even though it may be working on eight rows of C, it can only do so with four rows at a time (a cache line size). Each row can be thought of as an independent dot product, requiring a floating point stack location. Accesses to A are repeated each time a multiply is necessary, but fortunately we block A to be L1_data resident. Accesses to B, however, can be amortized over the four different dot products. Furthermore, to reduce the overhead of latency to L2, we typically keep two B's around, which in effect pre-touches the next B needed several cycles before it is first used. This effectively uses seven of the eight available floating point stack locations (four for the four dot products eventually going into C, two for B, and one for A loads). The whole length of all four dot products are unrolled to minimize overheads.

An unexpected benefit (about a five percent improvement) was observed by including the "fxch" floating point exchange instruction in selected points within our assembly DGEMM. Ironically, the fxch instructions were inserted in locations that did not impact the final result. That is, just before adding stack location 0 to 1, we would occasionally exchange stack location 0 and 1 first, thus adding stack location 1 to 0. Commutativity ensures these are the same, but apparently internal registers allocated to the tasks by the micro-operations treated the two situations somewhat differently. At one point we believed that the unnecessary fxchs were throttling the rate of the retiring instructions, bringing them in sync with the decoding instructions. But we also found that the spurious fxchs were only beneficial when data was running from cache. Around memory movements, taking some of the fxchs out again improved performance further. (This makes sense since something is more likely to occur around the large latency of a memory touch.) Although the fxch is supposed to be a "free" instruction, it takes up space in the reservation pool which has a limited capacity of 40 micro-operations. Exceeding this capacity leads to a stall.

We also used integer touches to pre-fetch C before it was needed. In effect, we would touch a cache line of C, do the 4 dot products, and then load C in to add it to the results.

Finally, when things were optimized on one processor, we split the matrix multiply up on two CPUs to maximize single node performance. Recall that a certain number of columns were blocked off of B and C to keep a strip of B in L2. The resulting matrix multiply was further stripped into groups of rows such that the relevant portion of A would remain inside the L1_data cache. We simply had one CPU take the odd group of rows and the other take the even so that both CPUs would be working on distinct, but close, portions of memory. Since B is not written to, having both CPUs share chunks of B in their respective L2 cache is not a problem.

Pieces of the DGEMM created for MP LINPACK were ported into the Intel Math Kernel Library currently available for Windows NT [5,7].

Other BLAS

MP LINPACK also relies partially on a matrix triangular solve with many right-hand sides. The upper triangular matrix is small (64x64), but the right-hand sides are large. We found that assembly tuning pieces of the upper triangular solve, interleaved with calls to DGEMM, yielded very high performances. The right-hand sides were split between the processors.

We are currently involved in providing UNIX-gnu-based optimized BLAS (and FFTs) for the Intel ASCI Option Red Supercomputer. But we also have efforts underway to provide extended precision math kernels. IA-32 naturally does work in 80-bit arithmetic. If we make an effort to directly support computation done in this framework (special IA-32 instructions exist for loading and storing 80-bit quantities to get around the 64-bit conversions), then some iterative codes might run faster. It is unlikely that the MFLOP rate will go up since doing 80-bit memory transactions is slower than their 64-bit counterparts (bus widths are usually 64-bits). However, the increased accuracy could enable less work to be done to ensure a final acceptance criterion, which would mean getting the answer faster. We are also looking into software-extended formats such as 160-bit arithmetic for this machine.

An Application Example

CTH is an Eulerian-Lagrangian code used at Sandia National Laboratories for shock physics studies. It contains approximately 440K lines of Fortran code, spread among ~1600 files. A parallel version of this code was developed for the Intel Paragon supercomputer at Sandia prior to the installation of the ASCI Option Red Supercomputer. Studies of this code showed that it scales nearly linearly with the number of computational nodes employed, suggesting that this code is appropriately balanced from the “massively parallel point of view.” This code is in continuous use on the Intel ASCI Option Red Supercomputer and has been run for extended periods (~150 hours) on a sizable number of nodes (2048), as well as having had a few limited runs on 4500 nodes, using over 100 Mbytes of the 128 Mbytes available per node. (It should be noted that the limiting factor on the duration of the full-machine runs was the machine schedule, rather than any hardware or software problems.) This is clearly a real application that can take advantage of the full system, and one where getting the optimal performance

has a real payoff. For example, a ten percent improvement would cut a 150 hour run down to 135 hours, shaving off over half a day, which is useful since management is often waiting on the answers and the queues for future runs are usually full.

Initial discussions with the CTH group revealed that the serial version of CTH, which runs on a wide variety of platforms, does not have well-defined kernels that could be tuned to provide significant speedup to the full code. Nevertheless, we examined a few of the most significant routines in order to spot repeated patterns that might be improved wholesale. The 3D EFP problem was chosen as a representative data-set for work on the CTH code. Profiling indicated that one of the most significant routines was ELSG, which is around 3700 lines of code. Using the ideas presented earlier in this paper, we investigated the performance of this routine.

We used the performance counters to gain an understanding of the memory characteristics of this code. This showed that the program's performance was bounded by the costs of memory movement, a surprising result given that the data appeared to be in the caches. More specifically, we found that the fraction of data from L1 was .85, the fraction from L2 was .12, and the fraction from memory was .03. Given the number of floating point operations based on the PP_FLOPS counter, this implied the maximum achievable performance for the particular data set was 27 MFLOPS. The actual performance was around 17 MFLOPS. Additional losses were due to branch misprediction, speculative execution, floating point dependencies, and other trouble spots.

We then took the major routine and created two instances of it, one for each CPU. Each CPU then did half the work with appropriate synchronizations being added to ensure correctness. The modified code ran slightly slower than the original code. Normalizing the original code's time to 1.0, the modified code ran at 1.13. The dual processor code ran at 0.61. Perfect speed-up was not possible because not every computation could be parallelized.

There were several other significant observations. We tried to avoid latency stalls associated with computing logical values by precomputing them, combining them, or removing them when possible. We used reciprocals when appropriate in order to minimize divides. We interlaced independent calculations to avoid floating point calculation stalls. We used the monitors to see where the data movement was going, and ended up achieving about 60-70 percent of the peak observable based on the memory movement.

One of the important functions of a supercomputer is the ability to run extremely large problems on an extremely large number of nodes reliably. CTH is an example of an application that has done just that. It has not only run on the full machine, but it has done so for a large number of uninterrupted hours.

Conclusions

The Intel ASCI Option Red Supercomputer is in routine production use. The machine is successfully addressing the problems that motivated the DOE to purchase it. One feature of the machine that we haven't talked about is its ability to rapidly switch between classified and unclassified operating modes [12]. While this isn't a performance issue, it does make the machine more broadly usable and therefore impacts the application programmer directly.

Performance on such a complex machine means many things. It means understanding single node performance, and knowing where the memory bottlenecks lie. In this paper, we have briefly discussed some of our more important findings in that area. It means understanding where the cycles are going for applications like CTH using tools such as the hardware counters. It means taking the care to do specialized tunings like asynchronous message passing and lookahead pivots to make codes like MP LINPACK parallelize well across a large number of nodes. It means experimenting with techniques like write combine memory to see when this is most beneficial. It means creating a performance suite to ensure that the compiler and the operating system are always running at optimal speeds.

Our performance and optimization studies are an ongoing effort. In this paper we have highlighted some of the major efforts and discoveries. Our final goal is to obtain correct codes running as fast as possible. We have demonstrated high theoretical peaks for important benchmarks like MP LINPACK. Application codes have been running on the machine for over a year now, even though we completed this supercomputer in June 1996.

Acknowledgements

Many people have worked on the MP LINPACK benchmark over the years. In addition to Greg Henry's work on the program, valuable contributions were made by Robert van de Geijn (University of Texas in Austin), Bob Norin (Intel Corp.) Brent Leback (Axian Corp.), Stuart Hawkinson (Axian Corp.), and Satya Gupta (Intel Corp.).

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., *LAPACK Users' Guide*, SIAM Publications, Philadelphia, PA, 1992.

[2] Blackford, S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S.,

Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C., *ScaLAPACK Users' Guide*, 1997, SIAM Publications, Philadelphia, PA 19104-2688, ISBN 0-89871-397-8.

[3] "Computers–Design Issues and Performance." Technical Paper in Supercomputing 1996, Proceedings of Supercomputing '96, Pittsburgh, Pennsylvania, http://www.supercomp.org/sc96/proceedings.

[4] Jerry Bolen, Arlin Davis, Bill Dazey, Satya Gupta, Greg Henry, David Robboy, Guy Schiffler, David Scott, Mack Stallcup, Amir Taraghi, Stephen Wheat, LeeAnn Fisk, Gabi Istrail, Chu Jong, Rolf Riesen, Lance Shuler, "Massively Parallel Distributed Computing: World's First 281 Gigaflop Supercomputer," Proceedings of the Intel Supercomputer Users Group 1995, http://www.cs.utk.edu/~ghenry/isug.ps.

[5] The Intel Performance Library suite, http://developer.intel.com/design/perftool/perflibst/

[6] Dongarra, J.J., "Performance of various computers using standard linear equations software in a Fortran environment," Computer Science Technical Report CS-89-85, University of Tennessee, 1989, http://www.netlib.org/benchmark/performance.ps

[7] Greer, B., Henry, G.,"High Performance Software on Intel Pentium Pro Processors or Micro-ops to TeraFlops," Proceedings for Supercomputing 1997, San Jose, CA.

[8] Gupta, S., Hawkinson, S., Henry, G., "Performance Of Matrix Matrix Multiply (DGEMM) For MP-LINPACK On Pentium Pro Processors," An Intel Internal Whitepaper, January 1996.

[9] Hendrickson, B.A., Womble, D.E., "The torus-wrap mapping for dense matrix calculations on massively parallel computers," SIAM J. Sci. Stat. Comput., 1994, http://www.cs.sandia.gov/~bahendr/torus.ps.Z

[10] S. Gupta and T.G. Mattson, "Optimization of MP QUEST for the ASCI Option Red System," Intel TFLOPS Project Research Report, 1996.

[11] Intel optimization manuals, http://developer.intel.com/design/pro/manuals/242816.htm and 242690.htm

[12] Mattson, T.G., and Henry, G, "The ASCI Option Red Supercomputer," Proceedings for ISUG 1997, Albuquerque, NM.

[13] Sears, M., *MP Quest User Guide*, Documentation distributed with the MP Quest program.

[14] Strassen, V., "Gaussian Elimination is not Optimal," Numer. Math. Vol. 13, 1969, pp. 354—356.

[15] van de Geijn, R.A., "Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC(R)/860 Systems," 1991 Annual Users' Conference Proceedings. Intel Supercomputer Users' Group, Dallas, TX, 10/91.

[16] Winograd, S., "A new algorithm for inner product," IEEE Trans. Comp., Vol. C-37, 1968, pp. 693—694.

[17] Pentium Pro Processor technical documents, http://www.intel.com/design/pro/manuals/.

Authors' Biographies

Greg Henry received his Ph.D. from Cornell University in Applied Mathematics. He started working at Intel SSD in August 1993. He is now a Computational Scientist for the ASCI Option Red Supercomputer. He tuned MP LINPACK and the BLAS used there-in. Greg has three children and a wonderful wife. He plays roller hockey, soccer, and he enjoys Aikido and writing. His e-mail is henry@co.intel.com

Pat Fay is presently an Intel computational scientist. He is responsible for assisting the Los Alamos National Laboratory scientists in using the Intel ASCI Option Red Supercomputer. He received his Ph.D. in Physics from Clemson University in 1993 and a Masters of International Business from the University of South Carolina in 1987. His e-mail is pfay@co.intel.com

Ben Cole is the Intel computational scientist on-site at Sandia National Laboratories. For his Ph.D. thesis, he studied transport processes in particle accelerators, comparing experimental results to a numerical model implemented on a parallel architecture. He has a second career as a father to an energetic three-year-old. His e-mail is cole@co.intel.com

Timothy G. Mattson has a Ph.D. in chemistry (1985, U.C Santa Cruz) for his research on Quantum Scattering theory. He has been with Intel since 1993 and is currently a research scientist in Intel's Parallel Algorithms Laboratory where he works on technologies to support the expression of parallel algorithms . Tim's life is centered on his family, snow skiing, science and anything that has to do with kayaks. His e-mail is timothy_g_mattson@ccm2.hf.intel.com.